Institut
Mines-Télécom

Introduction à OpenMP

Elisabeth Brunet
Télécom SudParis, septembre 2020

OpenMP - Plan

- **Introduction**
- **Modèle d'exécution**
- **Structure d'un programme OpenMP**
  - Construction d'une région parallèle
- **Portée des variables**
- **Parallélisme SPMD : Partage du travail et des données**
- **Exécutions exclusives**
- **Synchronisation au sein de régions parallèles**
- **Imbrication de régions parallèles**
- **Conclusion**

Introduction : Contexte

- **Architectures à mémoire partagée**
- **Exploitations possibles**
  - Plusieurs processus indépendants
  - Plusieurs processus collaborant via MPI par ex.
  - Parallélisation intrinsèque
    - Pthreads : librairie de threads POSIX
    - …
    - OpenMP : Open specifications for MultiProcessing

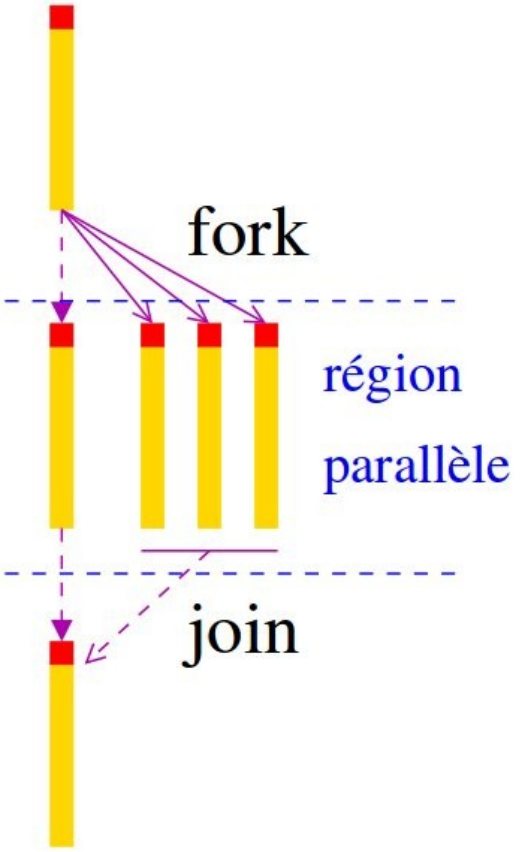
Introduction : Le standard OpenMP

- **Langage par pragmas permettant la description des régions parallèles dans du code écrit en séquentiel**
- **Support pour les langages C/C++ et Fortran**
- **Portable sur des nombreuses architectures et OS**

- **Standard dit industriel apparu en 1997**
- **Spécifications établies par l'ARB (Architecture Review Board) : http://openmp.org**
  - OpenMP2 en 2000 (extension pour Fortran 95),
  - OpenMP3 en 2008 (introduction des tâches),
  - OpenMP4 en 2013 (affinité et support des accélérateurs)

Modèle d'exécution

- **Modèle fork-join au sein d'un processus**
  - Région séquentielle par le thread maître
  - À chaque région parallèle, création d'une équipe de threads qui va se partager le travail
  - Synchronisation implicite et destruction des fils d'exécution parallèles à chaque fin de section parallèle



Structure d'un programme OpenMP : Pragmas OpenMP

- **Annotation du code grâce à l'ajout de pragmas OpenMP**
- **Syntaxe suivant le squelette :**
  - Sentinelle directive [clause[clause]...]
  - En C : sentinelle = #pragma omp
  - En fortran : sentinelle = $!OMP
- **Interprétation des pragmas si activée à la compilation, sinon cela reste des commentaires**
  - Portabilité ascendante des programmes

Structure d'un programme OpenMP : Construction d'une région parallèle

- **Directive parallel**
- **La seule directive qui lance la création des threads**
- **Dans la section parallèle, exécution redondante du code par chaque thread**

- **Attention !**
  - Bloc de code parallèle structuré : un seul point d'entrée et un seul point de sortie, ie pas de branchements vers l'intérieur ou l'exterieur du bloc
  - Valable pour toutes les constructions OpenMP

```
int main(){

    // code séquentiel

    #pragma omp parallel

    {

    /* code parallèle exécuté par
    tous les threads */

    }

    // code séquentiel

}
```

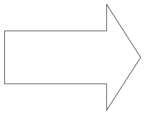
Structure d'un programme OpenMP : Librairie et environnement

- **Prototypes en C/C++ dans omp.h, en fortran dans OMP_LIB**
- **Définition de la taille de l'équipe de threads**
  - Variable d'environnement OMP_NUM_THREADS
  - Variation du nombre d'une construction parallèle à une autre
    - Clause num_threads de la directive parallel
    - Fonction omp_set_num_threads()
  - Choix du nombre optimal de threads par le runtime
    - variable d'environnement OMP_DYNAMIC positionnée à true
    - appel à la fonction omp_set_dynamic()
    - #max de threads utilisés = omp_get_max_threads()

Structure d'un programme OpenMP : Taille de l'équipe de threads

```
export OMP_NUM_THREADS = 4;
```

```
#pragma omp parallel
  printf(« Section 1\n»)

#pragma omp parallel num_threads(2)
  printf(« Section 2\n »)

omp_set_num_threads(3) ;
#pragma omp parallel
  printf(« Section 3\n»)
```



Section 1
Section 1
Section 1
Section 1
Section 2
Section 2
Section 3
Section 3
Section 3

Structure d'un programme OpenMP : Numérotation des threads

- **Le thread maître est numéroté 0**
- **Consultation**
  - Du nombre max de threads pouvant participer : omp_get_max_threads()
  - Du nombre de threads concurrents de la section : omp_get_num_threads()
  - De l'identifiant du thread : omp_get_thread_num()

Structure d'un programme OpenMP : Compilation et exécution

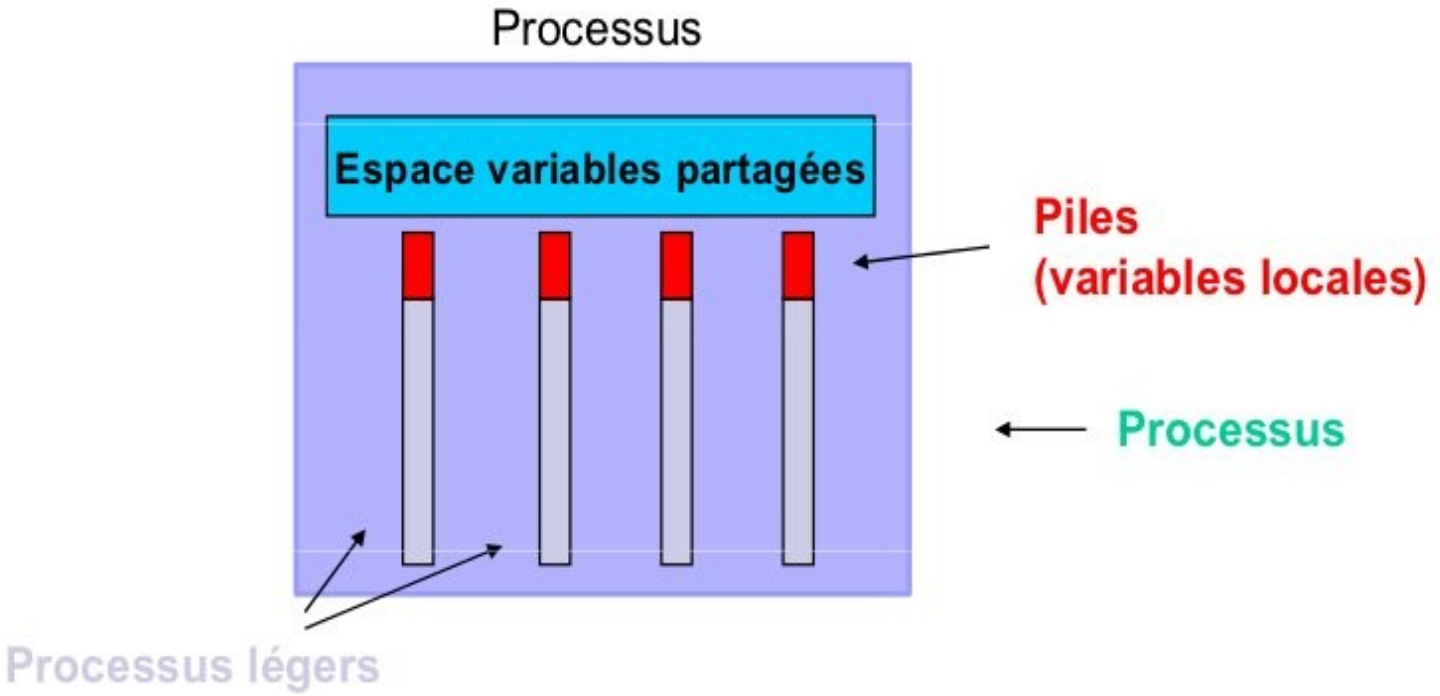
- **Compilation :**
  - Support dans la plupart des compilateurs classiques
  - Option à préciser à la compilation
  - En C : gcc -fopenmp prog.c -o prog
  - En fortran : gfortran -fopenmp -std=f95 prog.f
  - [Si compilation sans support OpenMP, utilisation de la macro _OPENMP afin de commenter les appels aux fonctions de la librairie OpenMP

```
#ifdef _OPENMP
   omp_get_num_threads()
#endif                    ]
```

- **Exécution**
  - Positionnement des variables d'environnement
    - OMP_NUM_THREADS, OMP_DYNAMIC, ...
  - Lancement classique : ./prog

Portée des variables

- **Accès usuel à la mémoire depuis les threads**
  - Accès à la mémoire virtuelle du processus
  - Accès à leur pile respective



Portée des variables :

Transmission aux sections parallèles

- **Par défaut, les variables sont shared**

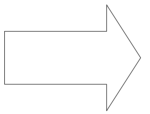
- **Clause shared**
  - Variable partagée qui reste en mémoire globale

- **Clause private**
  - Variable dupliquée dans chaque pile mais !! non initialisée !!
- **Clause firstprivate**
  - variable dupliquée dans chaque pile initialisée par la valeur de la variable d'origine

Portée des variables :

Transmission aux sections parallèles

```
omp_set_num_threads(2) ;
int a = 10, b = 20, c = 30 ;
#pragma omp parallel private (a) {
  printf(« %d \n», a + 1) ;
}
#pragma omp parallel  firstprivate (b){
  b = b+1 ;
  printf(« %d\n », b) ;
}
#pragma omp parallel shared (c){
  printf(« %d\n», c) ;
}
```



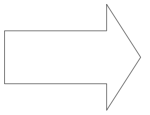
```
?
?
21
21
30
30
```

Portée des variables : Clause reduction

- **#pragma omp parallel reduction (op : liste)**
  - op : +, -, *, max, min, or, etc.
  - liste de variables scalaires partagées
- **Copie privée des variables initialisées selon l'opération de réduction donnée**
- **Réduction des copies locales à la fin de la section parallèle**

```
int res;
#pragma omp parallel reduction(+:res) {
  printf(« %d\n », omp_get_thread_num());
   res = res + omp_thread_num();
}
printf(« %d\n », res) ;
```

```
export OMP_NUM_THREADS = 3
```



```
0
1
2
3
```


TELECOM SudParis

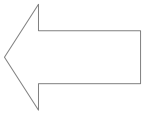
Portée des variables : Directive threadprivate

- **Fait persister une variable globale au programme mais privée à un thread d'une région parallèle à une autre**
- **Clause copyin affecte la variable à la valeur détenue par le thread 0 à tous les threads**

```
export OMP_NUM_THREADS = 3
```

```
int a = 10, b = 40;

#pragma omp threadprivate (a, b)
```

```
S1 - 2 - 2, ?
S1 - 0 - 0, 40
S1 - 1 - 1, ?
S2 – 2 – 2, 40
S2 – 1 – 1, 40
S2 – 0 - 0, 40
```

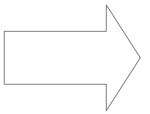


```
#pragma omp parallel private (tid)
{
  tid = omp_get_thread_num() ;
  a = tid ;
  printf(« S1 - %d – %d, %d\n », tid, a , b) ;
}
#pragma omp parallel private (tid) copyin(b)
{
  tid = omp_get_thread_num() ;
  printf(« S2 - %d – %d, %d\n », tid, a , b) ;
}
```

Portée des variables : Transmission par argument

- **Hérite de la portée définie pour la région parallèle courante**

```
void fonction (int a, int *b){
  *b = a + omp_get_thread_num();
}


int main(){
  int a = 5, b;
  omp_set_num_threads(6);

#pragma omp parallel private(b)
  {
    fonction(a, &b);
    printf("%d - a=%d, b=%d\n", omp_get_thread_num(), a, b);
  }
  return EXIT_SUCCESS;
}
```



```
3 - a=5, b=8
4 - a=5, b=9
0 - a=5, b=5
1 - a=5, b=6
2 - a=5, b=7
5 - a=5, b=10
```

Récapitulatif

- **Directive parallel**

```
#pragma omp parallel [clause ...]
                    if (scalar_expression)
                    private (list)
                    shared (list)
                    default (shared | none)
                    firstprivate (list)
                    reduction (operator: list)
                    copyin (list)
                    num_threads (integer)
{
  structured_block
}
```

- **Directive threaprivate**

```
#pragma omp threadprivate (list)
```

- **Spécification de la taille de l'équipe de threads**
- **Numérotation des threads**
- **Compilation/exécution**

Parallélisme SPMD : Partage du travail et des données

- **Contrôle fin de la répartition**
  - du calcul
  - des données traitées par chaque thread

- **Directives**
  - Boucle for
  - Exclusions mutuelles : directives master et single
  - Workshare (réservée au fortran)

Partage du travail : Directive for

- **Répartition des itérations d'une boucle suivant**
  - Ordonnancement par défaut défini à l'installation de l'environnement
  - Variable d'environnement OMP_SCHEDULE
  - Clause schedule
- **Portée des variables maintenue**
  - Clause lastprivate : conserve la valeur

    de la dernière itération au delà de la boucle
- **Clause nowait**
  - désactive la synchronisation en fin de région
- **Attention !**
  - Les indices de boucles sont entiers et privés
  - Pas de boucle infinie
  - Pas de boucle while

```
#pragma omp for [clause ...]
                schedule (type [,chunk])
                ordered
                collapse(n)
                private (list)
                firstprivate (list)
                lastprivate (list)
                shared (list)
                reduction (operator: list)
                nowait

  for_loop
```

- **Directive parallel for = fusion des constructions parallel et for**

Partage des données : Directive for - Clause schedule (1/2)

- **Spécification du mode de répartition des itérations**
- **Fonction omp_get_schedule(omp_sched_t *kind, int *modifier)**
  - kind={omp_sched_static|omp_sched_dynamic| omp_sched_guided|omp_sched_auto} et modifier = la taille d'itération positionnée
- **Clause schedule (static|dynamic|guided|runtime[, taille])**

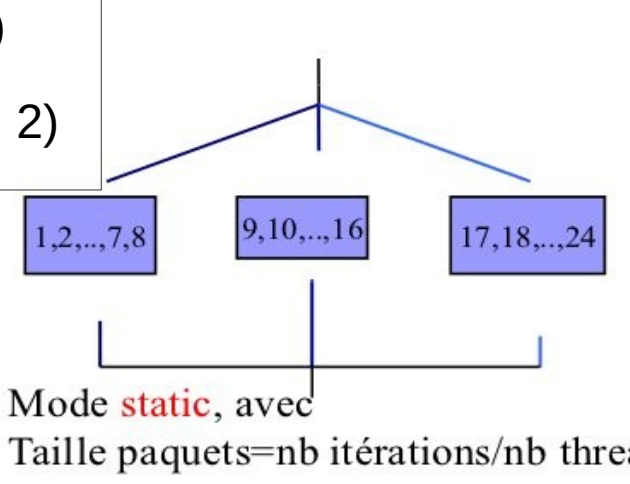
Partage des données : Directive for - Clause schedule (1/2)

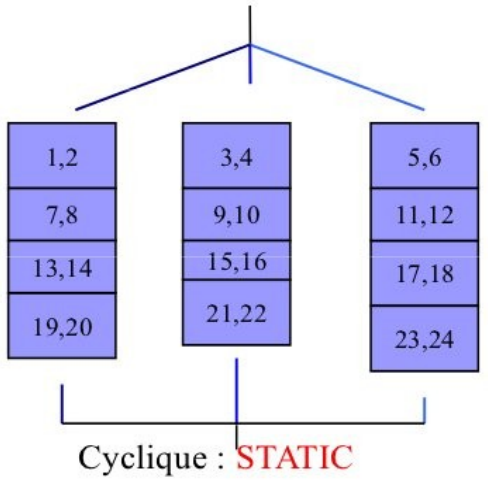
- **Spécification du mode de répartition des itérations**
- **Fonction omp_get_schedule(omp_sched_t *kind, int *modifier)**
  - kind={omp_sched_static|omp_sched_dynamic| omp_sched_guided|omp_sched_auto} et modifier = la taille d'itération positionnée
- **Clause schedule (static|dynamic|guided|runtime[, taille])**
  - **static** : divise les itérations par paquet de la taille donnée puis répartition cyclique
    - si non spécifiée, taille=nb_iter/nb_threads

Ex : 24 itérations, 3 threads

#pragma omp for schedule(static)

#pragma omp for schedule(static, 2)





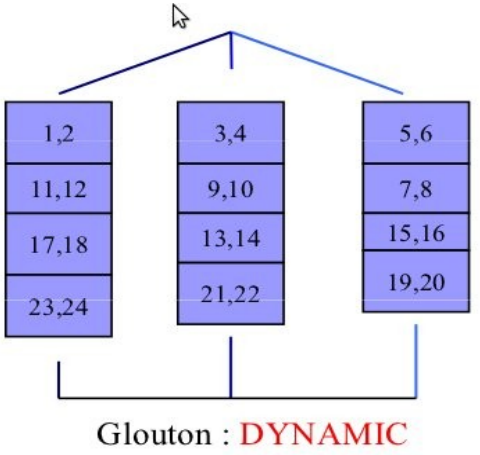
Partage des données : Directive for - Clause schedule (2/2)

- **dynamic** : les paquets de la taille donnée sont distribués aux threads libres de façon dynamique
  - Si non spécifiée, taille=1

Ex : 24 itérations, 3 threads

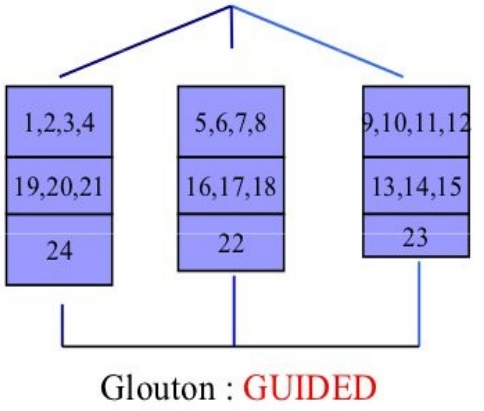
#pragma omp for schedule(dynamic,2)

#pragma omp for schedule(guided, 4)



- **guided** : les itérations sont divisées en paquets dont la taille décroît exponentiellement.
  - Si non spécifiée, taille=1

- **runtime** :
  - Choix différé au moment de l'exécution
  - Variable d'environnement OMP_SCHEDULE
  - Ex : export OMP_SCHEDULE= "dynamic, 2"



Partage du travail : Directive for - Clause collapse

- **Spécifie un nombre de boucle à déplier afin de créer un espace d'itérations plus large**

- **Permet d'augmenter le parallélisme**

```
#pragma omp parallel for collapse(2)
  for(i=0 ; i < M ; i++)
    for(j=0 ; j < N ; j++)
      for(k=0 ; k < K ; k++){
        fonc(i, j, k) ;
      }
```

- Attention !
  - Espace d'itération triangulaire interdit
  - Imbrication parfaite : pas d'opération intercalée entre les boucles

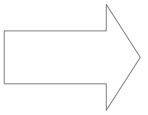
Sans le collapse :
- Répartition suivant l'indice i → max M threads
- Chaque thread exécute intégralement les 2 et 3ème boucles

Avec le collapse :
- Répartition suivant les indices i et j → max M*N threads
- Chaque thread exécute intégralement la 3ème boucle

Directive for - Clause ordered

- **Exécute la boucle séquentiellement**
  - Destinée au débuggage
  - Quelque soit la politique d'ordonnancement utilisée
  - Également disponible en tant que directive

```
A[4] = {1,2,3,4} ;
#pragma omp parallel for schedule(dynamic)
  for(i=0 ; i < 4 ; i++)
     printf(« B1 - %d\n », A[i]) ;

#pragma omp parallel for schedule(dynamic) ordered
  for(i=0 ; i < 4 ; i++)
     printf(« B2 - %d\n », A[i]) ;
```



```
B1 - 2
B1 - 3
B1 - 1
B1 - 4
B2 – 1
B2 – 2
B2 – 3
B2 - 4
```

Imbrication de constructions parallèles

- **Possibilité d'imbriquer des régions parallèles si**
  - Variable d'environnement OMP_NESTED positionnée
  - Appel à omp_set_nested()

```
#pragma omp parallel
{
  # pragma omp for
  for (i=0 ; i < 10 ; i++){
     ...
    #pragma omp parallel
    # pragma omp for
    for(j=0 ; j < 10 ; j++){
       ...
    }
  }
 }
```

- Attention !
  - L'imbrication induit l'utilisation d'une nouvelle région parallèle

Exécutions exclusives

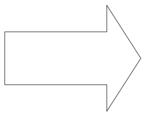
- **Directives**
  - Master
  - Single
  - Critical

Exécutions exclusives : Directive Master

- **Exécution par le thread maître uniquement**
- **Pas de synchronisation avec les autres threads**
  - Pas d'attente de début ou de fin d'exécution

```
#pragma omp master

  structured_block
```

```
int a, tid ;
#pragma omp parallel private (a, tid)
{
  a = 20 ;
  tid = omp_get_thread_num() ;

  #pragma omp master
  {
   a = 10 ;
  }

  printf(« %d, a=%d\n », tid, a) ;
}
```



```
1- 20
2 - 20
0 - 10
```

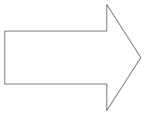
Exécutions exclusives : Directive Single

- **Exécution uniquement par le premier thread qui arrive à la construction**
- **Barrière implicite en fin de construction**
  - Threads ne participant pas attendent la fin de l'exécution avant de poursuivre leur exécution
  - Sauf si clause Nowait
- **Clause copyprivate**
  - Mise à jour des copies privées de tous les threads en fin de construction

```
#pragma omp single [clause ...]
                   private (list)
                   firstprivate (list)
                   copyprivate(list)
                   nowait

    structured_block
```

Exécutions exclusives : Illustration

```
int main(){
  int tid, a;

  omp_set_num_threads(4);

#pragma omp parallel private(tid, a)
  {
    tid = omp_get_thread_num();
    a = 888;

#pragma omp single
    a = 999;

    printf("%d - a = %d\n", tid, a);

#pragma omp master
    printf("%d – Fin de l'illustration\n", tid) ;
  }
  return EXIT_SUCCESS ;
}
```



```
0 - a = 888
3 - a = 888
2 - a = 999
1 - a = 888
0 – Fin de l'illustration
```

Exécutions exclusives : Directive critical

- **Accès en exclusion mutuelle à une portion de code**
- **Tous les threads participent à la construction mais un à la fois**
- **Ordre non déterministe**
  - ➔ premier arrivé, premier servi

```
#pragma omp critical [name]
  structured_block
```

Synchronisation

- **Directive barrier**

```
#pragma omp barrier
```

  - Tous les threads doivent avoir atteint la barrière avant de pouvoir
  - Barrière implicite à la fin des régions parallèles
    - Levée possible avec la clause nowait

- **Directive atomic**

```
#pragma omp atomic
  statement_expression
```

  - Mise à jour atomique en mémoire, ie en une seule opération
  - Modification de variable par un thread à la fois

- **Directive flush**

```
#pragma omp flush (list)
```

  - Garantit le rafraîchissement des variables partagées au niveau des threads
    - Variables en cache pour des questions de performances
    - Particulièrement utile sur les machines NUMA

Conclusion

- **OpenMP est un langage de haut niveau**
- **Permet de paralléliser rapidement un programme écrit en séquentiel**
- **Parallélisme orienté donnée**
- **Portée des variables**
- **Non traité :**
  - Parallélisme orienté tâches
  - Accélérateur, constructions simd, annulation, contrôle d'affinité de threads, redéfinition d'opérateur de réduction, sous-tableaux,...
- **Reference Card :**
  - openmp.org/mp-documents/OpenMP4.0-CCard.pdf